NONLINEAR OPTIMIZATION

Qingsha Cheng 程庆沙

Stochastic Search Methods II

Genetic algorithms: introduction

Representation

Genetic operators

Selection schemes

Selected topics

Simple implementation of a bit-string-based genetic algorithm

Genetic Algorithms

Genetic algorithms have been developed in late 1970s, mostly by J. Holland, K. DeJong and D. Goldberg

Initially, genetic algorithms processed populations of individuals represented by means of binary strings; nowadays, integer or floating point representation is preferred

Genetic algorithms traditionally emphasize the role of crossover

Genetic algorithms are considered to be good heuristics for combinatorial problems, although they may also used for continuous optimization

Heuristics

A heuristic technique (/hjuːˈrɪstɪk/; Ancient Greek: εὑρίσκω, "find" or "discover"), often called simply a heuristic, is any approach to problem solving, learning, or discovery that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals.

Examples of this method include using a rule of thumb, an educated guess, an intuitive judgment, guesstimate, stereotyping, profiling, or common sense.

heuristic vs. algorithm video

Genetic Algorithms

General structure of a genetic algorithm:

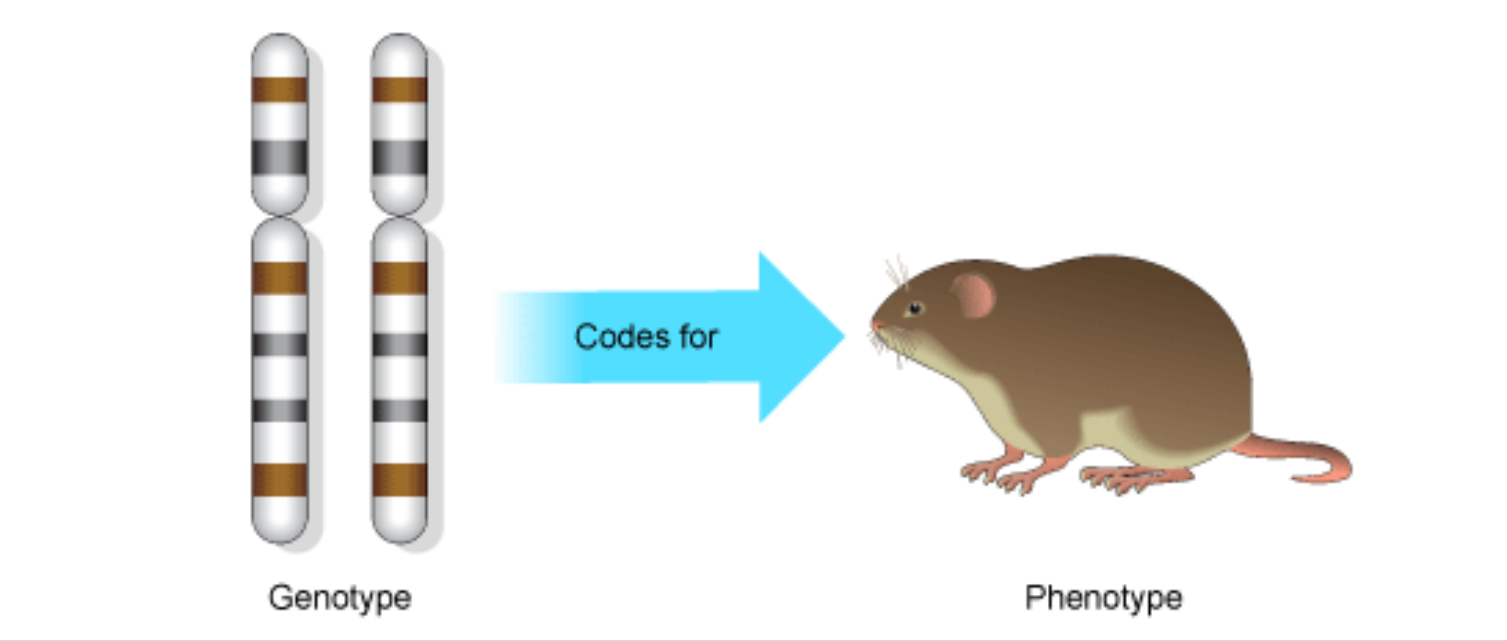
P = initialize_population(); % typically random initialization is preferred
F = evaluate_population(P); % assigns fitness value to each individual
while ~termination_condition
    P_{temp} = selection(P,F); % selects pairs of parents for the mating pool
    P_{new} = crossover(P_{temp},p_c); % crossover parents to create new individuals
    P_{new} = mutation(P_{new},p_m); % modify individuals using random changes
    P = P_{new}; % replace old population by a new one
    F = evaluate_population(P); % assigns fitness value to each individual
end

p_c is a crossover probability: each pair of parents undergoes crossover with probability p_c, otherwise, parents are copied

p_m is a mutation probability: bit-flip mutation is applied to each bit independently with probability p_m

Genetic Algorithms: Representation

Typically, genetic algorithms process individuals that are represented in certain internal format (*genotype*), recombination and mutation operations are performed on genotype; assessment of individual's fitness is performed on decoded individuals (*phenotype*)



Genetic Algorithms: Representation

Traditional genetic algorithms use bit-string representation, in particular, $x \in [a,b] \in R$ is represented by $\{c_1,\ldots,c_L\} \in \{0,1\}^L$; the function $g : \{0,1\}^L \rightarrow [a,b]$ defining the representation should be a one-to-one mapping, e.g.,

$$g(c_1,...,c_L) = a + \frac{b-a}{2^L - 1}\left(\sum_{j=0}^{L-1} 2^j c_{L-j}\right)$$

which is a standard binary encoding

L determines precision of the representation (high precision => long chromosomes => longer evolution)

Genetic Algorithms: Representation

Gray coding (named after Frank Gray)
a.k.a. reflected binary code (RBC) is often used because it is a binary numeral system where two successive values differ in only one bit.

It ensures the property:
small changes of the genotype imply small changes of the phenotype; this property makes genetic algorithms more efficient

Gray code by bit width

2-bit	4-bit
00	0000
01	0001
11	0011
10	0010
	0110
	0111
3-bit	0101
000	0100
001	1100
011	1101
010	1111
110	1110
111	1010
101	1011
100	1001
	1000

Genetic Algorithms: Representation

Currently, natural representation is preferred, e.g., floating point representation is used for continuous optimization problems, integer representation is used for combinatorial optimization (e.g., traveling salesman problem), etc.

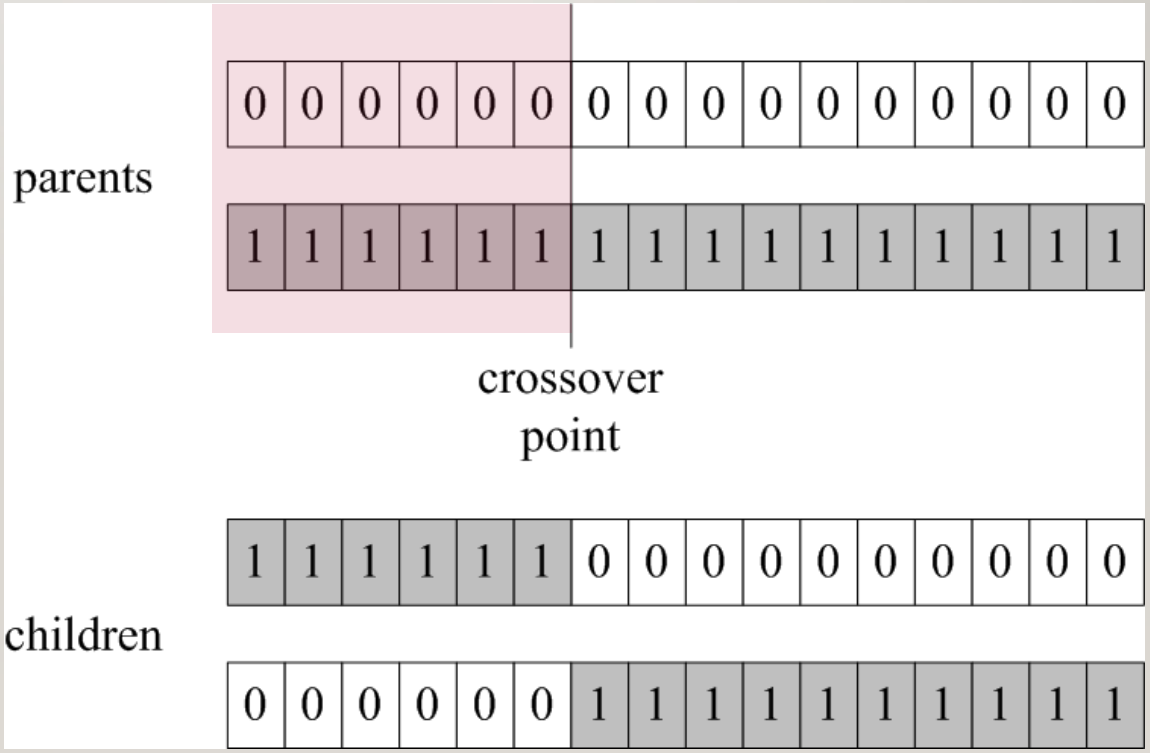
In many cases, representation is tailored to the problem in order to facilitate genetic operations or ensure feasibility of individuals

Genetic Algorithms: Crossover

Crossover is a basic operation in genetic algorithm designed to exchange the genetic information between parents

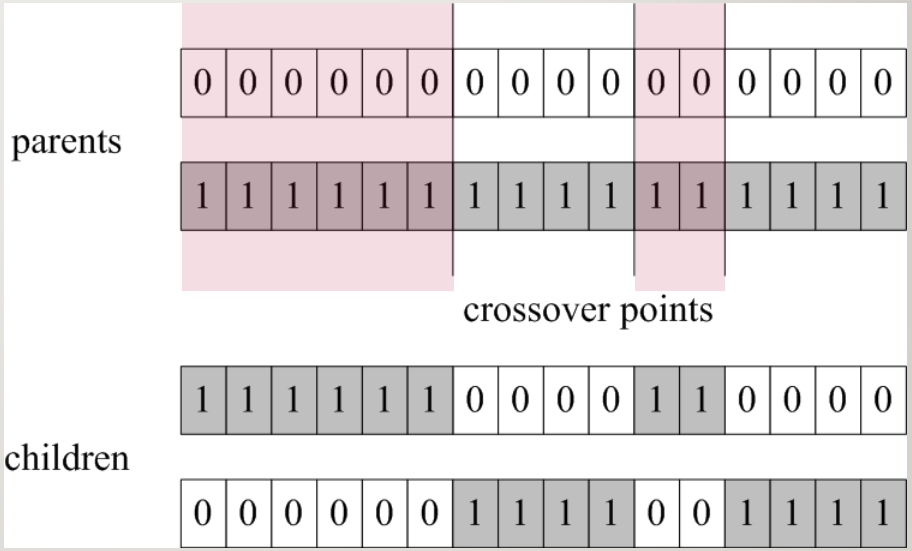
Examples of two-parent crossover operators for bit-string representation:

1. One-point crossover

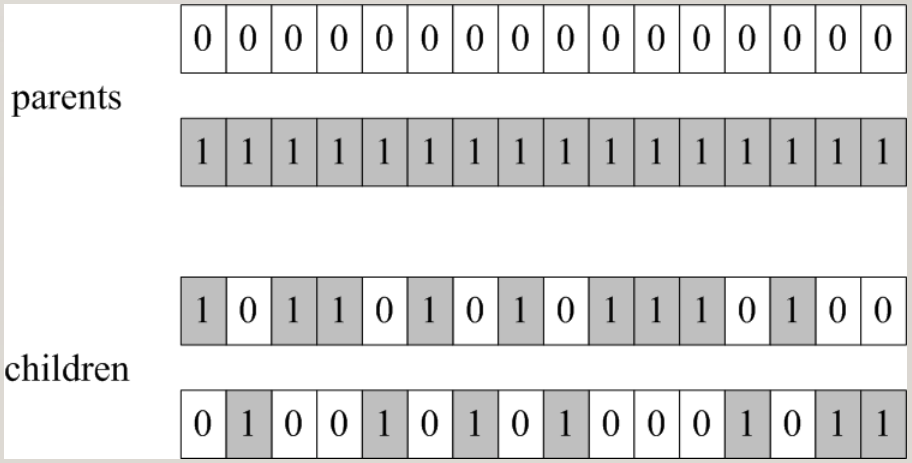


Genetic Algorithms: Crossover

2. Multi-point crossover



3. Uniform crossover (bits are exchanged randomly)



Genetic Algorithms: Crossover

Crossover is **explorative**, i.e., it makes large steps into an area somewhere in between the parents areas; therefore crossover helps discovering promising area in the search space and gaining information on the problem

Typical crossover rate, p_c, is high (e.g., 0.8), so that most of the parents exchange their data

Crossover may be generalized to involve more parents, however no significant advantages of this approach have been reported

Note: crossover does not produce new genetic information and, therefore, cannot be used as the only operator in genetic algorithm

Genetic Algorithms: Crossover

Crossover operators for floating point representation (assume that parents are $x = [x_1 \ldots x_n]^T$ and $y = [y_1 \ldots y_n]^T$, child is $z = [z_1 \ldots z_n]^T$):

1. Discrete

   $z_i = x_i$ or y_i (random selection)

2. Intermediate

   $z_i = ax_i + (1\text{-}a)y_i$ with $0 \leq a \leq 1$
   (parameter a selected randomly)

3. Arithmetic

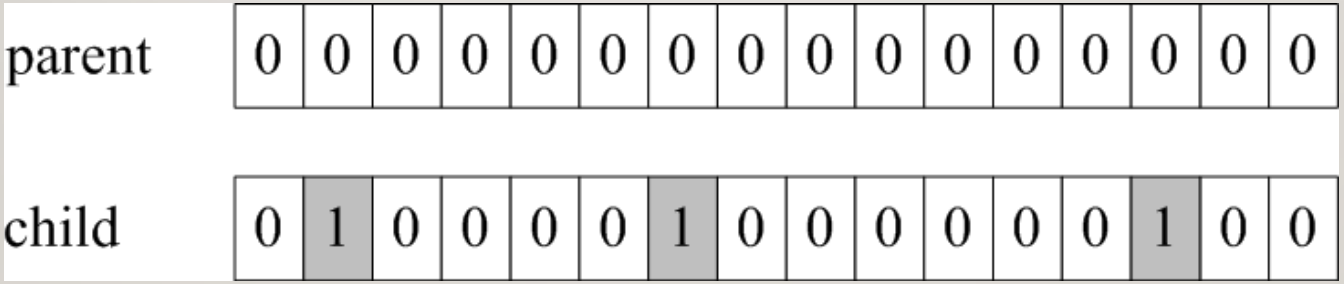
   $z = ax + (1\text{-}a)y$ with $0 \leq a \leq 1$ (parameter a selected randomly)

Variations and mixtures of the above operators are also used

Genetic Algorithms: Mutation

Mutation introduces small, random changes into the genotype

Mutation operator for bit-string representation: alter each gene independently with a probability p_m



Mutation rate, p_m, is normally small, typically between 1/population_size and 1/chromosome_length

Genetic Algorithms: Mutation

Mutation is **exploitative**, i.e., it creates random small changes, therefore mutation helps optimizing within a promising area using available information encoded in the chromosome

Mutation produces new genetic information and therefore, in principle, may be used as the only operator in a genetic algorithm (cf. early versions of evolution strategies)

However, it is good to use both mutation and crossover in general

Genetic Algorithms: Mutation

Mutation operator for floating point representation (assume that the individual $x = [x_1 \ldots x_n]^T$:

$$x_i \rightarrow x_i' = x_i + \Delta x_i$$

where Δx_i is a random deviation that may be drawn with uniform or non-uniform (e.g., normal) distribution;

Non-uniform distributions that promote smaller changes are typically preferred, e.g.,

$$\Delta x_i = \begin{cases} \left(x_{i.\max} - x_i\right) \cdot \left(2(r - 0.5)\right)^{\beta} & \text{if} \quad r > 0.5 \\ \left(x_{i.\min} - x_i\right) \cdot \left(2(0.5 - r)\right)^{\beta} & \text{otherwise} \end{cases}$$

where $r \in [0,1]$ is a random number; β is normally larger than 1 (e.g., 3) and may increase in later stages of the algorithm run

Genetic Algorithms: Selection

Selection is a process of choosing individuals that will become parents used to create the new population

Selection is based on the **fitness value** of individuals; normally the better individuals should get higher value of fitness (e.g., in case of minimization of the objective function f, the fitness of individual x could be $-f(x)$)

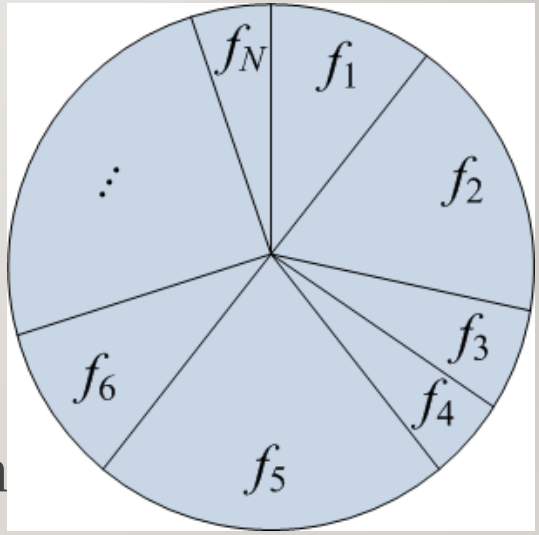
Selection promotes better individuals, however, it should contain an element of randomness to avoid premature convergence

video

Genetic Algorithms: Selection Schemes

Roulette wheel selection: expected number of copies of an individual i (i is an index of an individual) is

$$E(i) = N \cdot f_i / \langle f \rangle$$



where N is a population size
f_i is a fitness of individual i
$\langle f \rangle$ is an average fitness in the population

Implementation: draw a random number r in [0,1] with uniform distribution, then select individual i if

$$r \in \left[\frac{\sum_{k=1}^{i-1} f_k}{\sum_{k=1}^{N} f_k}, \frac{\sum_{k=1}^{i} f_k}{\sum_{k=1}^{N} f_k} \right)$$

Genetic Algorithms: Selection Schemes

Roulette wheel selection is the earliest selection scheme used in genetic algorithms and there is several problems associated with it, in particular:

1. A single highly-fit individual may take over the population
   => premature convergence

2. When all fitness values are similar (e.g., at the end of the algorithm run), the selection pressure is lost

Partial solution to problem 2: **fitness scaling**, e.g., use $f_i' = f_i - \beta$, where β is the worst fitness in the last iteration

Genetic Algorithms: Selection Schemes

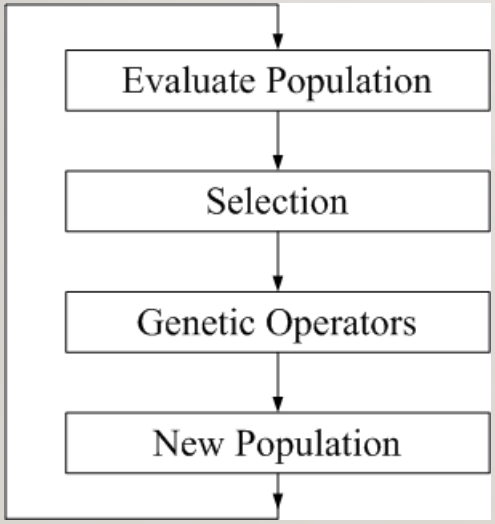
Tournament selection:
1.Randomly choose the subset of *k* individuals from the current population (*k* is a tournament size).
2.Select an individual that has the highest fitness value within the subset.
3.Repeat (steps 1 and 2) *N* times in order to create a parent set.

Tournament selection is free from problems of roulette wheel selection

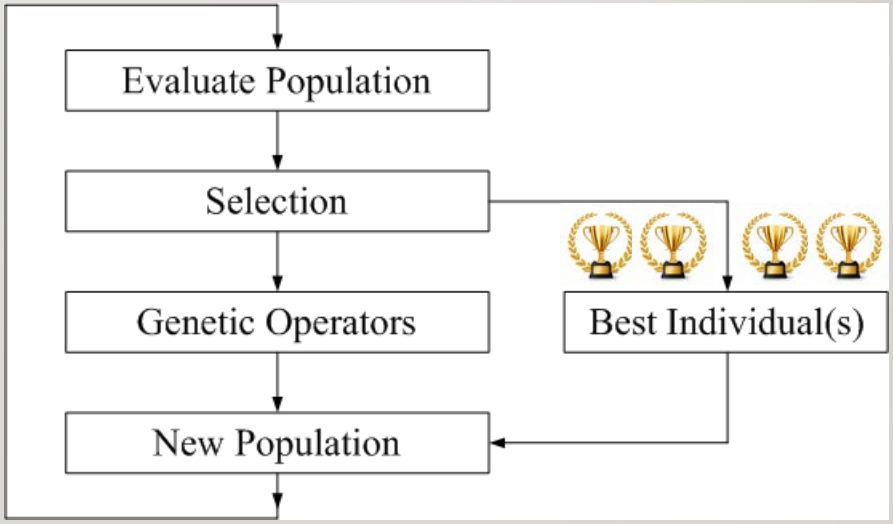
Tournament size determines the selection pressure (from pure random selection for $k = 1$, to fully deterministic selection for $k = N$; the latter would lead to instant convergence)

Genetic Algorithms: Selected Topics

Elitism: the technique frequently used in order to avoid loosing the best individual(s) found so far: the set of best individuals (typically one) is stored and deterministically inserted into the new population



Standard algorithm



Algorithm with elitism

Different versions of this technique have been considered; for example, there are algorithms where individuals may survive within the population up to a certain number of iterations

Genetic Algorithms: Selected Topics

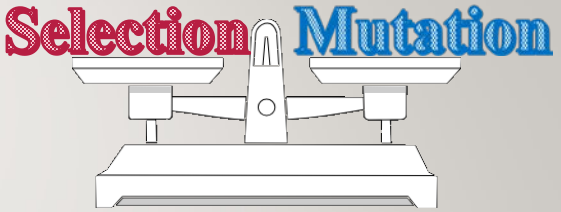
Population models: there are three basic approaches to replacing the old population by a new one:

1.Generational models: each individual survives for exactly one generation, and the entire set of parents is replaced by the offspring.

2.Steady-State models: one offspring is generated per iteration, and one member of the population is replaced.

3.Generation gap: the fraction of the population is replaced that may vary from 1.0 (as in generational model) to $1/N$ (as in steady-state models)

Genetic Algorithms: Selected Topics



Selection pressure, i.e., the **amount of preference** given to the better individuals to survive basically depends on the selection scheme used in an algorithm

Too high selection pressure may lead to premature convergence, i.e., the situation when all individuals become (almost) identical before the search space has been properly explored

Production of a new genetic material is a mechanism opposite to selection pressure, and it is mainly dependent on a mutation rate

Normally, we want to keep a **balance** between the **selection pressure** and **production of a new genetic material** so that the premature convergence is avoided but the evolution is not a random search either

Genetic Algorithms: Selected Topics

Practical methods for controlling the **balance** between **selection pressure** and **the production of a new genetic material** are based on monitoring diversity of the population and adjusting mutation rate accordingly (self-adaptation)

Diversity of the population may be measured **using standard deviation of individuals** (details depend on a particular representation used in the algorithm)

Mutation rate can be **reduced at the final stages** of the algorithm run in order to improve exploitation at the expense of exploration and, therefore, obtain more precise localization of the optimal solution

Genetic Algorithms: Broader View

Genetic algorithms are considered to be **special case** of a broader class of methods called **evolutionary algorithms**; in fact genetic algorithm using floating-point representation are considered by some to be evolutionary algorithms but not genetic algorithms

In general, evolutionary algorithms are distinguished by:

1. The use of population
2. The use of stochastic search operators
3. The use of stochastic selection

Genetic Algorithms: Theory

There is a little of theory explaining why evolutionary algorithms work

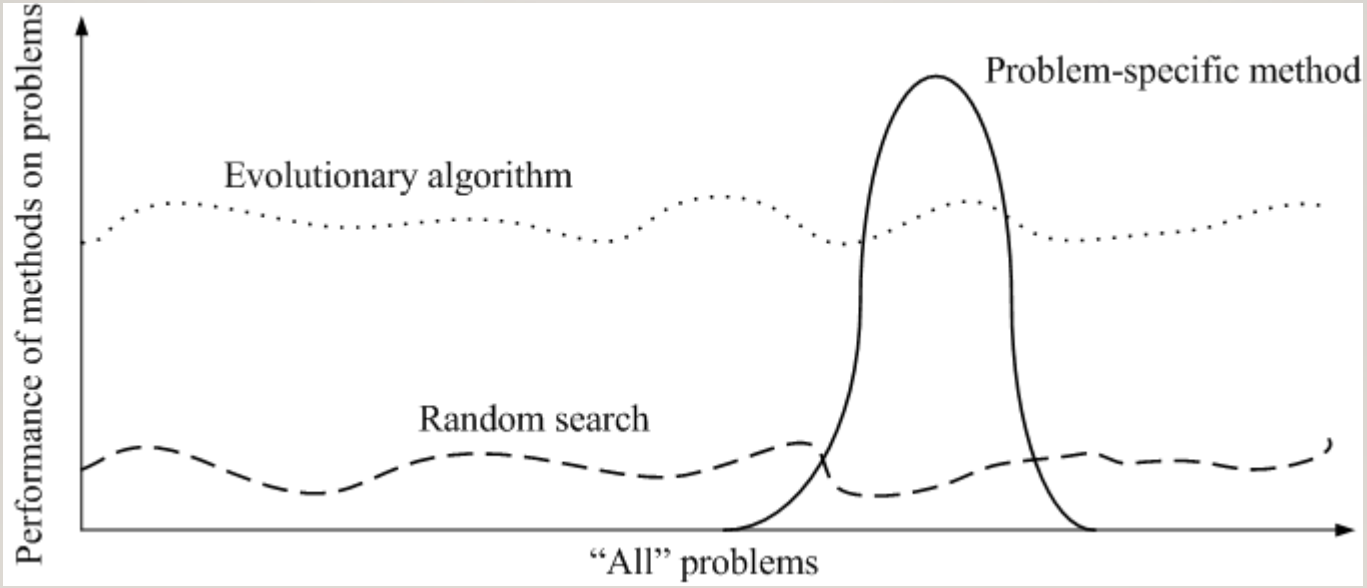
Early, semi-heuristic results for bit-string-based genetic algorithms (so-called schemata theorem by J. Holland) suggesting that genetic algorithms work by discovering, exploiting and combining “building blocks” (groups of closely interacting genes), have been later criticized (in particular, because of success of algorithms using natural representations)

There are results using Markov Chain analysis (where genetic algorithms are treated as stochastic processes) giving almost sure convergence of algorithm to a global solution; practical significance of such results is minor

Evolutionary Algorithms versus Other Methods

Evolutionary algorithms are considered robust problem solvers in the following sense:

1. They do not perform as good as problem-specific methods for problems for which the latter methods were designed.
2. They perform better than a random search “on average”.
3. They perform better than problem-specific methods “on average”.



Evolutionary Algorithms: Current Trends

It is recommended that evolutionary algorithms should be **individually designed** for solving a given problem

Adding problem-specific knowledge to the algorithm is essential to obtain good performance (e.g., by using suitable representation and tailored operators)

Constraint satisfaction should be ensured using repair algorithms and suitable genetic operators that **preserve feasibility of solutions**

Search for an “all-purpose” method is considered to be fruitless

Example: Implementation of Bit-String-Based Genetic Algorithm

Main function (arguments: *fun* – objective function handle, *N* – population size, *lb*, *ub* – lower and upper bounds for design variables, *pc*, *pm* – crossover and mutation probability, *L* – chromosome length, *k_max* – maximum number of generations)

```
function [xbest,fbest] = sga(fun,N,lb,ub,pc,pm,L,k_max)
pm0 = pm;
P = initialize_population(length(lb),N,L);   %% P is the poluation
[F,xbest,fbest] = evaluate(fun,P,lb,ub);      %% F is the fitness of the individuals in P
while k < k_max
    Ptemp = selection(P,F);
    P = crossover(Ptemp,pc);
    P = mutation(P,pm);
    P{1} = xbest;
    [F,xbest,fbest] = evaluate(fun,P,lb,ub,fbest);
    pm = adjust_mutation_rate(P,k,k_max,pm,pm0);
    k = k+1;
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Initialization: initial population is selected randomly, assuming uniform probability distribution.

```
function P = initialize_population(n,N,L)
for j = 1:N
    P{j} = floor(2*rand(1,n*L));
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Evaluation: individuals are evaluated after being decoded (i.e., bit-string representation is transformed into floating point one); best individual in the population is also found

```
function [F,xbest,fbest] = evaluate(fun,P,lb,ub,varargin)
j0 = 1; fbest = Inf;
if nargin>4
    j0 = 2;
    xbest = P{1};
    fbest = varargin{1};
    F(1) = fbest;
end
for j = j0:N
    F(j) = feval(decode_bin(P{j},lb,ub));
    if F(j) < fbest
        xbest = P{j};
        fbest = F(j);
    end
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Selection: the selection function produces $2N$ parent individuals using binary tournament selection (choose 2 each time and select the better one)

```
function R = selection(P,F)
N = length(P);
for j = 1:2*N
    c = ceil(N*rand(1,2));     %% random values [c1 c2]  ci=1~N
    if F(c(1)) < F(c(2))
        R{j} = P{c(1)};
    else
        R{j} = P{c(2)};
    end
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Mutation: bit-flip random mutation is implemented; mutation is applied to each individual gene with the probability p_m

```
function P = mutation(P,pm)
N = length(P);
n = length(P{1});
for j = 1:N                    %% Go through all genes in each individual for the
    for k = 1:n                %% entire population
        if rand() < pm
            P{j}(k) = 1 – P{j}(k);    %% flip the kth binary number
    end
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Crossover: uniform crossover is implemented and applied to each pair of parent individuals with probability p_m; if crossover is not performed for a given pair, one of the parents is taken as an offspring

```
function R = crossover(P,pc)
N = length(P);
n = length(P{1});
for j = 1:N/2
    if rand() < pc
        c = floor(2*rand(1,n));    %% P{2*j} and P(2*j-1} are a pair of parents
        for k = 1:n
            R{j}(k) = P{2*j-c(k)}(k);
        end
    else
        R{j} = P{2*j};
    end
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Adaptive mutation: if $k < k_{max}/2$, mutation rate is adjusted based on the current diversity of population (see the next slide); otherwise, mutation rate is decreasing according to the formula below

```
function pm = adjust_mutation_rate(P,k,k_max,pm,pm0)
S = population_diversity(P);   %% check population diversity
if k < k_max/2
    if S < 0.1
        pm = pm*1.3;
    else
        pm = pm/1.2;
    end
else
    pm = pm0*(2*(k_max-k)/k_max)^2;
end
```

Example: Implementation of Bit-String-Based Genetic Algorithm

Population diversity: calculates diversity of population measured as an average standard deviation of gene values

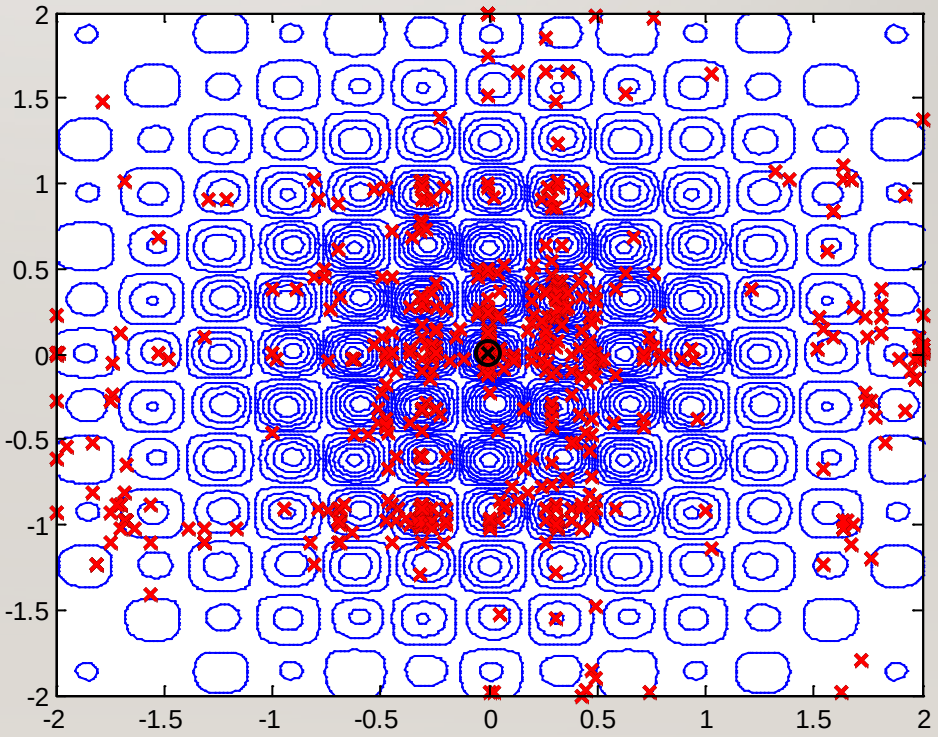
```
function S = population_diversity(P)
N = length(P);
n = length(P{1});
for j = 1:n
    for k = 1:N
        L(k) = P{k}(j);
    end
    R(j) = std(L);    %% standard deviation of jth gene
end
S = sum(R)/n;
```

Example: Bit-String Genetic Algorithm for Minimizing f_p

Setup:
1. Bit-string representation with Gray coding (10 bits/variable)
2. Population size $N = 10$, tournament selection with elitism
3. Uniform crossover ($p_c = 0.8$), mutation rate $p_m = 0.03$
4. Population-diversity-based self-adaptation of p_m

Minimum function value found: -0.9938

Total number of function evaluations: 1000



Example: Bit-String Genetic Algorithm Minimizing f_p

Statistics of 20 runs of the bit-string-based genetic algorithm minimizing the function f_p

Best Result	Average Result	Worst Result	Standard Deviation
-0.9938	-0.9199	-0.8061	0.060

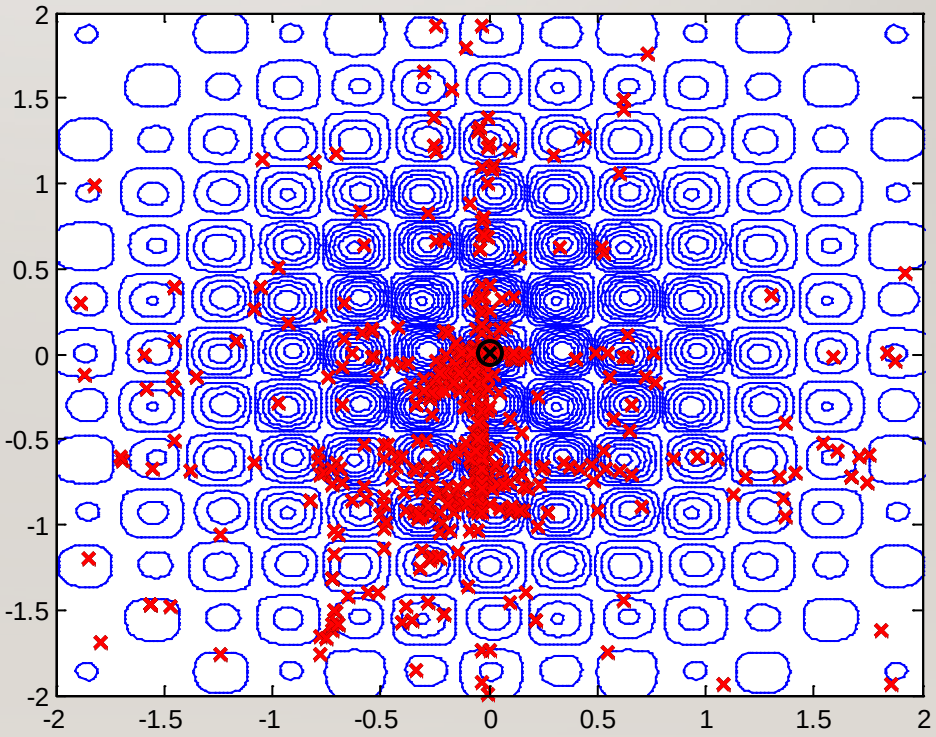
Remark: performance is better multiple-run gradient-search and random search but not as good as simulated annealing nor evolution strategies

Example: Floating-Point Genetic Algorithm for Minimizing f_p

Setup: 1. Floating-point representation
2. Population size $N = 10$, tournament selection with elitism
3. Arithmetic crossover ($p_c = 0.8$), mutation rate $p_m = 0.2$
4. Population-diversity-based self-adaptation of p_m

Minimum function value found: -0.9991

Total number of function evaluations: 1000



Example: Floating-Point Genetic Algorithm Minimizing f_p

Statistics of 20 runs of the floating-point-based genetic algorithm minimizing the function f_p

Best Result	Average Result	Worst Result	Standard Deviation
-1.0000	-0.9579	-0.9059	0.047

Remark: genetic algorithm with floating-point representation is better than the algorithm using bit-string representation but still not as good as evolution strategies nor simulated annealing

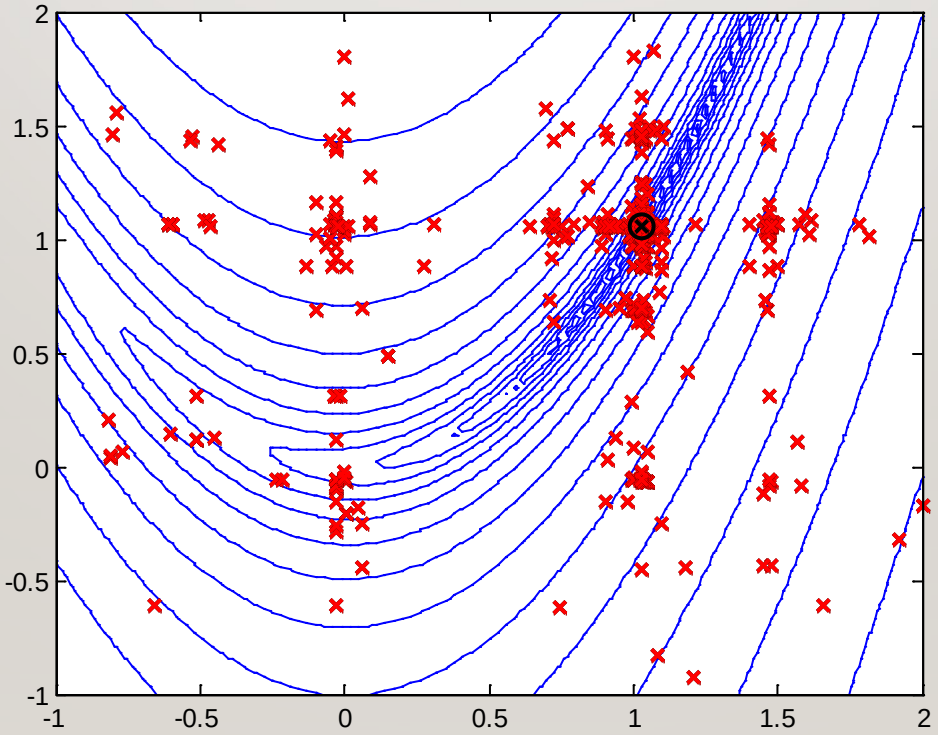
Comparison of Stochastic Search Methods: Function f_p

Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search	-0.9997	-0.8457	-0.6099	0.129
Random search	-0.9948	-0.9028	-0.7852	0.060
"Smart" random search	-1.0000	-0.9951	-0.9061	0.021
Simulated annealing	-1.0000	-0.9710	-0.9061	0.043
Evolution strategies	-1.0000	-0.9957	-0.9522	0.011
Particle swarm optimization	-1.0000	-0.9538	-0.8255	0.055
Bit-string GA	-0.9938	-0.9199	-0.8061	0.060
Floating point GA	-1.0000	-0.9579	-0.9059	0.047

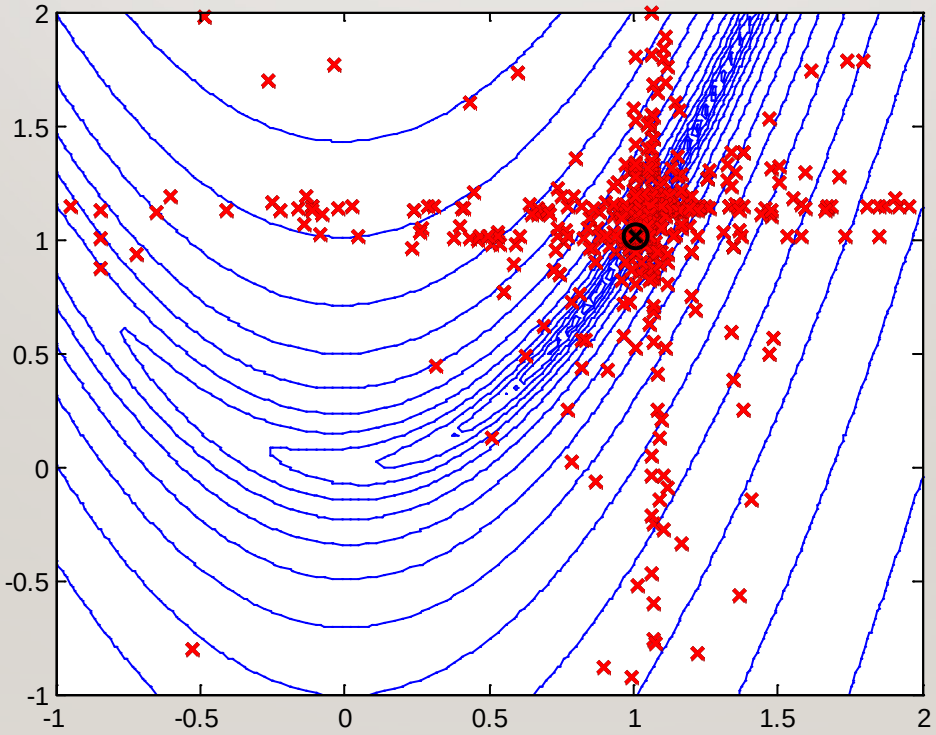
Example: Bit-String Genetic Algorithm for Minimizing Rosenbrock Function

Typical search pattern:



Example: Floating-Point Genetic Algorithm for Minimizing Rosenbrock Function

Typical search pattern:



Comparison of Stochastic Search Methods: Rosenbrock Function

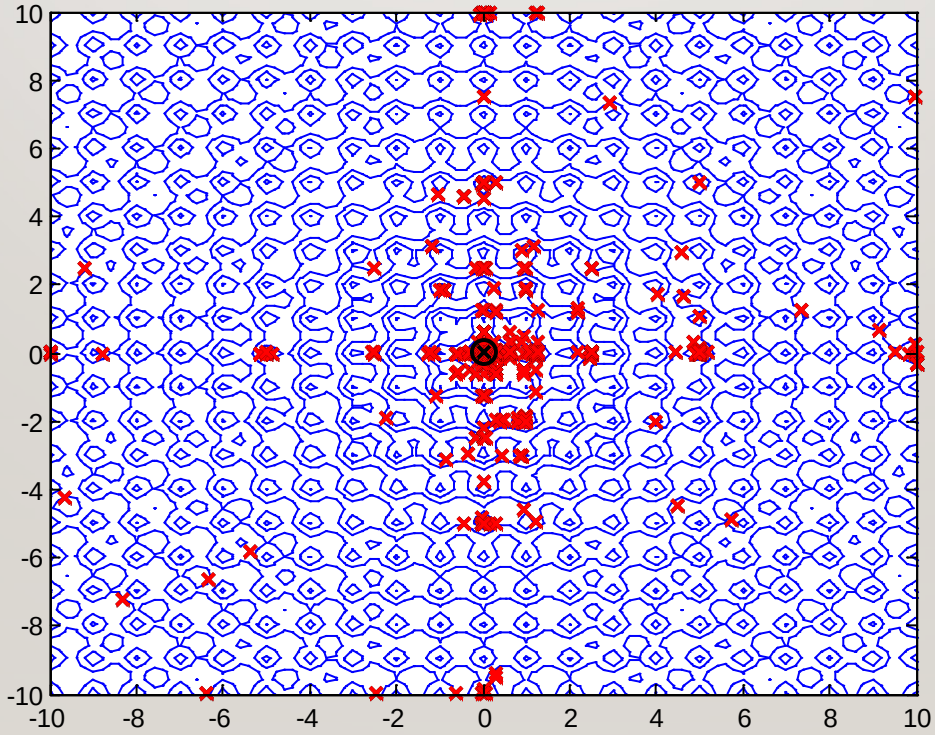
Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search*	$2.4 \cdot 10^{-6}$ (0*)	$8.2 \cdot 10^{-2}$ (0*)	$3.1 \cdot 10^{-2}$ (0*)	$1.0 \cdot 10^{-2}$ (0*)
Random search	$4.5 \cdot 10^{-5}$	$3.7 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$
"Smart" random search	$6.2 \cdot 10^{-7}$	$1.5 \cdot 10^{-4}$	$1.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-4}$
Simulated annealing	$5.0 \cdot 10^{-6}$	$8.5 \cdot 10^{-4}$	$5.6 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$
Evolution strategies	$3.9 \cdot 10^{-7}$	$4.7 \cdot 10^{-3}$	$4.3 \cdot 10^{-2}$	$9.7 \cdot 10^{-3}$
Particle swarm optimization	$2.8 \cdot 10^{-7}$	$5.1 \cdot 10^{-3}$	$2.3 \cdot 10^{-2}$	$7.7 \cdot 10^{-3}$
Bit-string GA	$5.8 \cdot 10^{-4}$	$1.5 \cdot 10^{-1}$	$7.3 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$
Floating point GA	$1.9 \cdot 10^{-7}$	$4.9 \cdot 10^{-2}$	$1.6 \cdot 10^{-1}$	$6.1 \cdot 10^{-2}$

* The single-start algorithm always finds a global minimum with sufficient number of function evaluations

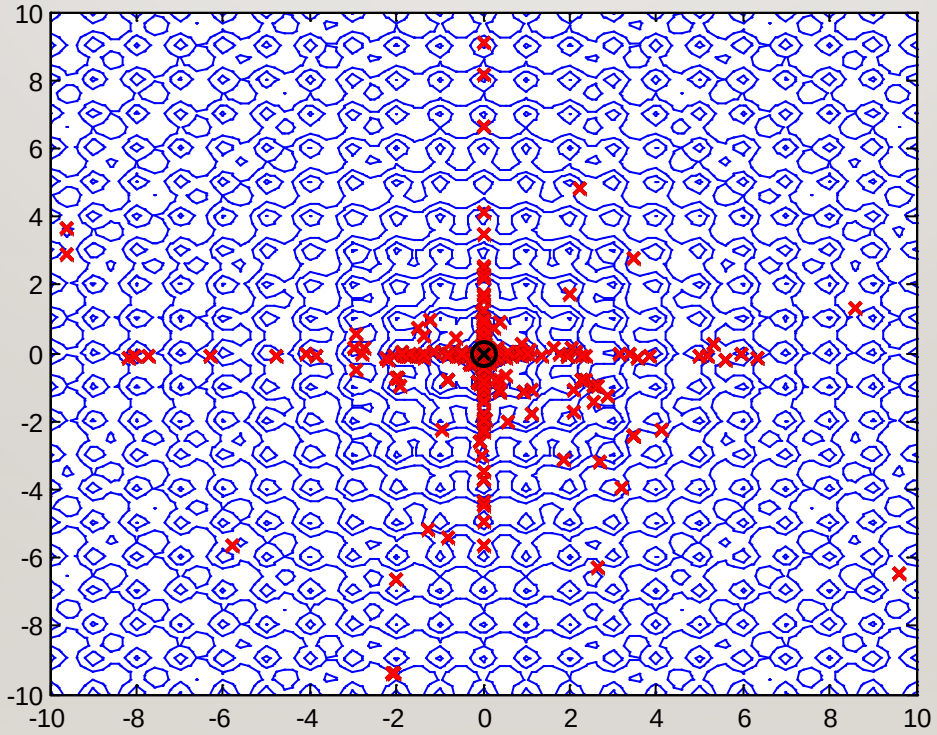
Example: Bit-String Genetic Algorithm for Minimizing Auckley Function

Typical search pattern:



Example: Floating-Point Genetic Algorithm for Minimizing Auckley Function

Typical search pattern:



Comparison of Stochastic Search Methods: Auckley Function

Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search	-1.7182	1.7244	5.4661	2.17
Random search	-1.4596	0.2036	1.1830	0.72
"Smart" random search	-1.7106	-1.6835	-1.6496	0.02
Simulated annealing	-1.7172	-1.7132	-1.7054	0.004
Evolution strategies	-1.7183	-1.7181	-1.7153	0.0007
Particle swarm optimization	-1.7183	-1.7182	-1.7178	0.0001
Bit-string GA	-1.7177	-1.7043	-1.6200	0.02
Floating point GA	-1.7173	-1.7026	-1.5400	0.04

Comparison of Stochastic Search Methods: Function f_p (n = 5)

Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search	-0.6135	-0.3354	-0.1055	0.143
Random search	-0.4073	-0.2859	-0.1351	0.069
"Smart" random search	-0.9056	-0.7276	-0.4138	0.146
Simulated annealing	-0.9046	-0.4298	-0.1234	0.204
Evolution strategies	-1.0000	-0.7295	-0.4150	0.131
Particle swarm optimization	-0.9069	-0.6857	-0.3764	0.176
Bit-string GA	-0.8222	-0.5732	-0.3357	0.134
Floating point GA	-0.9900	-0.8097	-0.5042	0.121

Comparison of Stochastic Search Methods: Rosenbrock Function ($n = 5$)

Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search*	0.0330 (0*)	0.5274 (0*)	1.4186 (0*)	0.473 (0*)
Random search	7.1031	14.8165	21.3041	3.488
"Smart" random search	0.0030	1.8780	4.5043	1.613
Simulated annealing	0.0019	2.6610	4.5100	1.686
Evolution strategies	0.6380	2.1327	4.5314	1.027
Particle swarm optimization	0.0692	2.0929	4.8601	1.765
Bit-string GA	0.4390	5.0856	52.9287	11.335
Floating point GA	0.0056	2.1119	4.6213	1.244

* The single-start algorithm always finds a global minimum with sufficient number of function evaluations

Comparison of Stochastic Search Methods: Auckley Function ($n = 5$)

Statistics based on 20 runs of each algorithm

Algorithm	Best Result	Average Result	Worst Result	Standard Deviation
Multiple-start gradient-based search	2.6796	6.9066	9.7872	2.045
Random search	2.3630	4.5634	6.0501	0.938
"Smart" random search	-1.6426	-0.8469	1.1754	0.959
Simulated annealing	-1.7065	-0.9635	0.6013	0.859
Evolution strategies	-1.7183	-1.6024	0.5986	0.518
Particle swarm optimization	-1.7183	-1.7183	-1.7182	0.00001
Bit-string GA	-1.7036	-1.5536	-0.7942	0.020
Floating point GA	-1.7089	-1.6347	-1.0985	0.133

Bibliography

D.E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, Reading, MA, 1989.

Z. Michalewicz, *Genetic algorithms + data structures = evolution programs*, 3rd edn, Springer, New York, 1996.

T. Back, D.B. Fogel, and Z. Michalewicz (Editors), *Evolutionary computation 1: basic algorithms and operators*, Taylor & Francis Group, 2000.

D.B. Fogel, *Evolutionary computation: toward a new philosophy of machine intelligence*, IEEE Press, 2006.

Researchers at SUSTech

YAO, Xin Department Head

Email: xiny_AT_sustc.edu.cn

Office: Room 908, Building A7, Nanshan i-Park

Research Area: Evolutionary Computation; Computational Intelligence; Machine Learning; Data Science; Meta-heuristic Optimisation; Search-based Software Engineering

SHI, Yuhui Chair Professor

Email: shiyh_AT_sustc.edu.cn

Office: Room 1011, Building A7, Nanshan i-Park

Research Area: Brain Storm Optimization Algorithms; Particle Swarm Optimization Algorithms; Swarm Intelligence; Evolutionary Computation, Machine Learning; Computational Intelligence; Artificial Intelligence; Data Science; Intelligent System

Researchers at SUSTech

ISHIBUCHI, Hisao Chair Professor

Email: hisao_AT_sustc.edu.cn

Office: Room 902, Building A7, Nanshan i-Park

Research Area: Computational Intelligence, Evolutionary Multiobjective Optimization, Evolutionary Machine Learning, Fuzzy Systems, Linguistic Data Mining, Evolutionary Games

TANG, Ke Prof.

Email: tangk3_AT_sustc.edu.cn

Office: Room 905, Building A7, Nanshan i-Park

Research Area: Computational Intelligence, Evolutionary Computation, Machine Learning

Exercise 1: Genetic Algorithms with Floating-Point Representation

Implement genetic algorithm using floating-point representation.
Use self-adaptive mutation rate.

Test the algorithm on:
(1)Rosenbrock function (n = 2, $-2 \leq x_i \leq 2$),
(2)function f_p (n = 1, 2, $-2 \leq x_i \leq 2$),
(3)Auckley's function (n = 1, 2 and 3, $-10 \leq x_i \leq 10$).

Perform some experiments to check how the performance of the algorithm depends on its control parameters (e.g., N, p_m, p_c, etc.)

Exercise 2: Evolutionary Algorithm for TSP

There are N cities with the distance between city i and city j given by d_{ij}. Problem: find a close route through all the cities so that minimizes the total route length.

Develop and implement an evolutionary algorithm for TSP.

1. Representation: permutation of the numbers 1 to N.
2. Mutation: random swapping of two cities on the route.
3. Crossover: develop an operator that maintains feasibility of individual (i.e., an individual created by crossover is still a valid permutation of integers 1 to N).
4. Adaptive mutation rate: monitor population diversity and use it to adjust mutation rate following the general rules discussed during the lecture.

Test the algorithm on randomly generated cases of the sizes 20 to 500. Implement some sort of visualization. Compare the results with the simple random swapping algorithm developed on Day 2.

AI Plays a Game